Section 2: Fortran terms and concepts

2.1 High level syntax

This section introduces the terms associated with program units and other Fortran concepts above the construct, statement, and expression levels and illustrates their relationships. The notation used in this standard is described in 1.6.

NOTE 2.1

Some of the syntax rules in this section are subject to constraints that are given only at the appropriate places in later sections.

R201 *program* **is** *program-unit*
  [*program-unit*] ...

A *program* shall contain exactly one *main-program program-unit*.

R202 *program-unit* **is** *main-program*
  or *external-subprogram*
  or *module*
  or *block-data*

R1101 *main-program* **is** [*program-stmt*]
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-program-stmt

R203 *external-subprogram* **is** *function-subprogram*
  or *subroutine-subprogram*

R1216 *function-subprogram* **is** *function-stmt*
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-function-stmt

R1221 *subroutine-subprogram* **is** *subroutine-stmt*
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-subroutine-stmt

R1104 *module* **is** *module-stmt*
  [*specification-part*]
  [*module-subprogram-part*]
  end-module-stmt

R1112 *block-data* **is** *block-data-stmt*
  [*specification-part*]
  end-block-data-stmt

R204 *specification-part* **is** [*use-stmt*] ...
  [*implicit-part*]
  [*declaration-construct*] ...

R205	*implicit-part*	**is**	[*implicit-part-stmt*] ... *implicit-stmt*
R206	*implicit-part-stmt*	**is**	*implicit-stmt*
		or	*parameter-stmt*
		or	*format-stmt*
		or	*entry-stmt*
R207	*declaration-construct*	**is**	*derived-type-def*
		or	*interface-block*
		or	*type-declaration-stmt*
		or	*specification-stmt*
		or	*parameter-stmt*
		or	*format-stmt*
		or	*entry-stmt*
		or	*stmt-function-stmt*
R208	*execution-part*	**is**	*executable-construct* [*execution-part-construct*] ...
R209	*execution-part-construct*	**is**	*executable-construct*
		or	*format-stmt*
		or	*entry-stmt*
		or	*data-stmt*
R210	*internal-subprogram-part*	**is**	*contains-stmt* *internal-subprogram* [*internal-subprogram*] ...
R211	*internal-subprogram*	**is**	*function-subprogram*
		or	*subroutine-subprogram*
R212	*module-subprogram-part*	**is**	*contains-stmt* *module-subprogram* [*module-subprogram*] ...
R213	*module-subprogram*	**is**	*function-subprogram*
		or	*subroutine-subprogram*
R214	*specification-stmt*	**is**	*access-stmt*
		or	*allocatable-stmt*
		or	*common-stmt*
		or	*data-stmt*
		or	*dimension-stmt*
		or	*equivalence-stmt*
		or	*external-stmt*
		or	*intent-stmt*
		or	*intrinsic-stmt*
		or	*namelist-stmt*
		or	*optional-stmt*
		or	*pointer-stmt*
		or	*save-stmt*
		or	*target-stmt*
R215	*executable-construct*	**is**	*action-stmt*
		or	*case-construct*
		or	*do-construct*
		or	*forall-construct*
		or	*if-construct*

or *where-construct*

R216 *action-stmt* **is** *allocate-stmt*
or *assignment-stmt*
or *backspace-stmt*
or *call-stmt*
or *close-stmt*
or *continue-stmt*
or *cycle-stmt*
or *deallocate-stmt*
or *endfile-stmt*
or *end-function-stmt*
or *end-program-stmt*
or *end-subroutine-stmt*
or *exit-stmt*
or *forall-stmt*
or *goto-stmt*
or *if-stmt*
or *inquire-stmt*
or *nullify-stmt*
or *open-stmt*
or *pointer-assignment-stmt*
or *print-stmt*
or *read-stmt*
or *return-stmt*
or *rewind-stmt*
or *stop-stmt*
or *where-stmt*
or *write-stmt*
or *arithmetic-if-stmt*
or *computed-goto-stmt*

Constraint: An *execution-part* shall not contain an *end-function-stmt*, *end-program-stmt*, or *end-subroutine-stmt*.

2.2 Program unit concepts

Program units are the fundamental components of a Fortran program. A **program unit** may be a main program, an external subprogram, a module, or a block data program unit. A subprogram may be a function subprogram or a subroutine subprogram. A module contains definitions that are to be made accessible to other program units. A block data program unit is used to specify initial values for data objects in named common blocks. Each type of program unit is described in Sections 11 or 12. An **external subprogram** is a subprogram that is not in a main program, a module, or another subprogram. An **internal subprogram** is a subprogram that is in a main program or another subprogram. A **module subprogram** is a subprogram that is in a module but is not an internal subprogram.

A program unit consists of a set of nonoverlapping scoping units. A **scoping unit** is

(1) A derived-type definition (4.4.1),

(2) A procedure interface body, excluding any derived-type definitions and procedure interface bodies in it (12.3.2.1), or

(3) A program unit or subprogram, excluding derived-type definitions, procedure interface bodies, and subprograms in it.

A scoping unit that immediately surrounds another scoping unit is called the **host scoping unit**.

2.2.1 Program

A **program** consists of exactly one main program unit and any number (including zero) of other kinds of program units. The set of program units may include any combination of the different kinds of program units in any order as long as there is only one main program unit.

NOTE 2.2

There is a restriction that there shall be no more than one unnamed block data program unit (11.4).

Since the public portions of a module are required to be available by the time a module reference (11.3.1) is processed, a processor may require a specific order of processing of the program units.

2.2.2 Main program

The main program is described in 11.1.

2.2.3 Procedure

A **procedure** encapsulates an arbitrary sequence of computations that may be invoked directly during program execution. Procedures are either functions or subroutines. A **function** is a procedure that is invoked in an expression; its invocation causes a value to be computed which is then used in evaluating the expression. The variable that returns the value of a function is called the **result variable**. A **subroutine** is a procedure that is invoked in a CALL statement or by a defined assignment statement (12.4, 12.4.3, 7.5.1.3). Unless it is a pure procedure, a subroutine may be used to change the program state by changing the values of any of the data objects accessible to the subroutine; unless it is a pure procedure, a function may do this in addition to computing the function value.

Procedures are described further in Section 12.

2.2.3.1 External procedure

An **external procedure** is a procedure that is defined by an external subprogram or by means other than Fortran. An external procedure may be invoked by the main program or by any procedure of a program.

2.2.3.2 Module procedure

A **module procedure** is a procedure that is defined by a module subprogram (R213). A module procedure may be invoked by another module subprogram in the module or by any scoping unit that accesses the module procedure by use association (11.3.2). The module containing the subprogram is called the **host** of the module procedure.

2.2.3.3 Internal procedure

An **internal procedure** is a procedure that is defined by an internal subprogram (R211). The containing main program or subprogram is called the **host** of the internal procedure. An internal procedure is local to its host in the sense that the internal procedure is accessible within the scoping units of the host and all its other internal procedures but is not accessible elsewhere.

2.2.3.4 Procedure interface block

The purpose of a procedure **interface block** is to describe the interfaces (12.3) to a set of procedures and to optionally permit them to be invoked through either a single generic name, a defined operator, or a defined assignment. It determines the forms of reference through which the procedures may be invoked (12.4).

2.2.4 Module

A **module** contains (or accesses from other modules) definitions that are to be made accessible to other program units. These definitions include data object declarations, type definitions, procedure definitions, and procedure interface blocks. The purpose of a module is to make the definitions it contains accessible to all other program units that request access. A scoping unit in another program unit may request access to the definitions in a module. Modules are further described in Section 11.

2.3 Execution concepts

Each Fortran statement is classified as either an executable statement or a nonexecutable statement. There are restrictions on the order in which statements may appear in a program unit, and certain executable statements may appear only in certain executable constructs.

2.3.1 Executable/nonexecutable statements

Program execution is a sequence, in time, of computational actions. An **executable statement** is an instruction to perform or control one or more of these actions. Thus, the executable statements of a program unit determine the computational behavior of the program unit. The executable statements are all of those that make up the syntactic class of *executable-construct*.

Nonexecutable statements do not specify actions; they are used to configure the program environment in which computational actions take place. The nonexecutable statements are all those not classified as executable. All statements in a block data program unit shall be nonexecutable. A module is permitted to contain executable statements only within a subprogram in the module.

Table 2.1 Requirements on statement ordering

<table>
<tr><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE,
MODULE, or BLOCK DATA statement</td></tr>
<tr><td colspan="3">USE statements</td></tr>
<tr><td rowspan="4">FORMAT
and
ENTRY
statements</td><td colspan="2">IMPLICIT NONE</td></tr>
<tr><td>PARAMETER
statements</td><td>IMPLICIT
statements</td></tr>
<tr><td>PARAMETER
and DATA
statements</td><td>Derived-type definitions,
interface blocks,
type declaration statements,
specification statements,
and statement function statements</td></tr>
<tr><td>DATA
statements</td><td>Executable
constructs</td></tr>
<tr><td colspan="3">CONTAINS statement</td></tr>
<tr><td colspan="3">Internal subprograms
or module subprograms</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

2.3.2 Statement order

The syntax rules of subclause 2.1 specify the statement order within program units and subprograms. These rules are illustrated in Table 2.1 and Table 2.2. Table 2.1 shows the ordering rules for statements and applies to all program units and subprograms. Vertical lines delineate varieties of statements that may be interspersed and horizontal lines delineate varieties of statements that shall not be interspersed. Internal or module subprograms shall follow a

CONTAINS statement. Between USE and CONTAINS statements in a subprogram, nonexecutable statements generally precede executable statements, although the ENTRY statement, FORMAT statement, and DATA statement may appear among the executable statements. Table 2.2 shows which statements are allowed in a scoping unit.

Table 2.2 Statements allowed in scoping units

Kind of scoping unit:	Main program	Module	Block data	External subprog	Module subprog	Internal subprog	Interface body
USE statement	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ENTRY statement	No	No	No	Yes	Yes	No	No
FORMAT statement	Yes	No	No	Yes	Yes	Yes	No
Misc. declarations (see note)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DATA statement	Yes	Yes	Yes	Yes	Yes	Yes	No
Derived-type definition	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface block	Yes	Yes	No	Yes	Yes	Yes	Yes
Executable statement	Yes	No	No	Yes	Yes	Yes	No
CONTAINS statement	Yes	Yes	No	Yes	Yes	No	No
Statement function statement	Yes	No	No	Yes	Yes	Yes	No

Notes for Table 2.2:
1) Misc. declarations are PARAMETER statements, IMPLICIT statements, type declaration statements, and specification statements.
2) Derived type definitions are also scoping units, but they do not contain any of the above statements, and so have not been listed in the table.
3) The scoping unit of a module does not include any module subprograms that the module contains.

2.3.3 The END statement

An *end-program-stmt*, *end-function-stmt*, *end-subroutine-stmt*, *end-module-stmt*, or *end-block-data-stmt* is an **END statement**. Each program unit, module subprogram, and internal subprogram shall have exactly one END statement. The *end-program-stmt*, *end-function-stmt*, and *end-subroutine-stmt* statements are executable, and may be branch target statements. Executing an *end-program-stmt* causes termination of execution of the program. Executing an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a *return-stmt* in a subprogram.

The *end-module-stmt* and *end-block-data-stmt* statements are nonexecutable.

2.3.4 Execution sequence

Execution of a program begins with the first executable construct of the main program. The execution of a main program or subprogram involves execution of the executable constructs within its scoping unit. When a procedure is invoked, execution begins with the first executable construct appearing after the invoked entry point. With the following exceptions, the effect of execution is as if the executable constructs are executed in the order in which they appear in the main program or subprogram until a STOP, RETURN, or END statement is executed. The exceptions are the following:

(1) Execution of a branching statement (8.2) changes the execution sequence. These statements explicitly specify a new starting place for the execution sequence.

(2) CASE constructs, DO constructs, and IF constructs contain an internal statement structure and execution of these constructs involves implicit internal branching. See Section 8 for the detailed semantics of each of these constructs.

(3) END=, ERR=, and EOR= specifiers may result in a branch.

(4) Alternate returns may result in a branch.

Internal subprograms may precede the END statement of a main program or a subprogram. The execution sequence excludes all such definitions.

2.4 Data concepts

Nonexecutable statements are used to define the characteristics of the data environment. This includes typing variables, declaring arrays, and defining new data types.

2.4.1 Data type

A **data type** is a named category of data that is characterized by a set of values, together with a way to denote these values and a collection of operations that interpret and manipulate the values. This central concept is described in 4.1.

There are two categories of data types: intrinsic types and derived types.

2.4.1.1 Intrinsic type

An **intrinsic type** is a type that is defined implicitly, along with operations, and is always accessible. The intrinsic types are integer, real, complex, character, and logical. The properties of intrinsic types are described in 4.3. An intrinsic type may be parameterized, in which case the set of data values depends on the values of the parameters. Such a parameter is called a **type parameter** (4.3). The type parameters are KIND and LEN.

The **kind type parameter** indicates the decimal exponent range for the integer type (4.3.1.1), the decimal precision and exponent range for the real and complex types (4.3.1.2, 4.3.1.3), and the representation methods for the character and logical types (4.3.2.1, 4.3.2.2). The **character length parameter** specifies the number of characters for the character type.

2.4.1.2 Derived type

A **derived type** is a type that is not defined implicitly but requires a type definition to declare components of intrinsic or of other derived types. A scalar object of such a derived type is called a **structure** (5.1.1.7). The only intrinsic operation for derived types is assignment with type agreement (7.5.1.5). For each derived type, structure constructors are available to provide values (4.4.4). In addition, data objects of derived type may be used as procedure arguments and function results, and may appear in input/output lists. If additional operations are needed for a derived type, they shall be supplied as procedure definitions.

Derived types are described further in 4.4.

2.4.2 Data value

Each intrinsic type has associated with it a set of values that a datum of that type may take. The values for each intrinsic type are described in 4.3. Because derived types are ultimately specified in terms of components of intrinsic types, the values that objects of a derived type may assume are determined by the type definition and the sets of values of the intrinsic types.

2.4.3 Data entity

A **data entity** is a data object, the result of the evaluation of an expression, or the result of the execution of a function reference (called the function result). A data entity has a data type (either intrinsic or derived) and has, or may have, a data value (the exception is an undefined variable). Every data entity has a rank and is thus either a scalar or an array.

2.4.3.1 Data object

A **data object** (often abbreviated to **object**) is a constant (4.1.2), a variable (6), or a subobject of a constant. The type of a named data object may be specified explicitly (5) or implicitly (5.3).

Subobjects are portions of certain named objects that may be referenced and defined (variables only) independently of the other portions. These include portions of arrays (array elements and array sections), portions of character strings (substrings), and portions of structures (components). Subobjects are themselves data objects, but subobjects are referenced only by subobject designators. A subobject of a variable is a variable. Subobjects are described in Section 6.

Objects referenced by a name are:

a named scalar	(a scalar object)
a named array	(an array object)

Subobjects referenced by a subobject designator are:

an array element	(a scalar subobject)
an array section	(an array subobject)
a structure component	(a scalar or an array subobject)
a substring	(a scalar subobject)

2.4.3.1.1 Variable

A **variable** may have a value and may be defined and redefined during execution of a program.

2.4.3.1.2 Constant

A **constant** has a value and cannot become defined or redefined during execution of a program. A constant with a name is called a **named constant** and has the PARAMETER attribute (5.1.2.1). A constant without a name is called a **literal constant** (4.3).

2.4.3.1.3 Subobject of a constant

A **subobject of a constant** is a portion of a constant. The portion referenced may depend on the value of a variable.

NOTE 2.3

For example, given:

```
CHARACTER (LEN = 10), PARAMETER :: DIGITS = '0123456789'
CHARACTER (LEN = 1)             :: DIGIT
INTEGER :: I
   ...
DIGIT = DIGITS (I:I)
```

DIGITS is a named constant and DIGITS (I:I) designates a subobject of the constant DIGITS.

2.4.3.2 Expression

An **expression** (7.1) produces a data entity when evaluated. An expression represents either a data reference or a computation, and is formed from operands, operators, and parentheses. The type, value, and rank of an expression result are determined by the rules in Section 7.

2.4.3.3 Function reference

A **function reference** (12.4.2) produces a data entity when the function is executed during expression evaluation. The type and rank of a function result are determined by the interface of the function (12.2.2). The value of a function result is determined by execution of the function.

2.4.4 Scalar

A **scalar** is a datum that is not an array. Scalars may be of any intrinsic type or derived type.

NOTE 2.4

A structure is scalar even if it has arrays as components.

The **rank** of a scalar is zero. The shape of a scalar is represented by a rank-one array of size zero.

2.4.5 Array

An **array** is a set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. An **array element** is one of the individual elements in the array and is a scalar. An **array section** is a subset of the elements of an array and is itself an array.

An array may have up to seven dimensions, and any **extent** (number of elements) in any dimension. The **rank** of the array is the number of dimensions, and its **size** is the total number of elements, which is equal to the product of the extents. An array may have zero size. The **shape** of an array is determined by its rank and its extent in each dimension, and may be represented as a rank-one array whose elements are the extents. All named arrays shall be declared, and the rank of a named array is specified in its declaration. The rank of a named array, once declared, is constant and the extents may be constant also. However, the extents may vary during execution for a dummy argument array, an automatic array, a pointer array, and an allocatable array.

Two arrays are **conformable** if they have the same shape. A scalar is conformable with any array. Any intrinsic operation defined for scalar objects may be applied to conformable objects. Such operations are performed element-by-element to produce a resultant array conformable with the array operands. Element-by-element operation means corresponding elements of the operand arrays are involved in a "scalar-like" operation to produce the corresponding element in the result array, and all such element operations may be performed in any order or simultaneously. Such an operation is described as **elemental**.

A rank-one array may be constructed from scalars and other arrays and may be reshaped into any allowable array shape (4.5).

Arrays may be of any intrinsic type or derived type and are described further in 6.2.

2.4.6 Pointer

A **pointer** is a variable that has the POINTER attribute. A pointer is **associated** with a target by allocation (6.3.1) or pointer assignment (7.5.2). A pointer shall neither be referenced nor defined until it is associated. A pointer is **disassociated** following execution of a DEALLOCATE or NULLIFY statement, following pointer association with a disassociated pointer, or initially through pointer initialization. A disassociated pointer is not currently associated with a target (14.6.2). If the pointer is an array, the rank is declared, but the extents are determined when the pointer is associated with a target.

2.4.7 Storage

Many of the facilities of this standard make no assumptions about the physical storage characteristics of data objects. However, program units that include storage association dependent features shall observe certain storage constraints (14.6.3).

2.5 Fundamental terms

The following terms are defined here and used throughout this standard.

2.5.1 Name and designator

A **name** is used to identify a program constituent, such as a program unit, named variable, named constant, dummy argument, or derived type. The rules governing the construction of names are given in 3.2.1. A **subobject designator** is a name followed by one or more of the following: component selectors, array section selectors, array element selectors, and substring selectors.

2.5.2 Keyword

The term **keyword** is used in two ways in this standard. A word that is part of the syntax of a statement is a **statement keyword**. These keywords are not reserved words; that is, names with the same spellings are allowed. Examples of statement keywords are: IF, READ, UNIT, KIND, and INTEGER.

An **argument keyword** is a dummy argument name (12.4). Section 13 specifies argument keywords for all of the intrinsic procedures. Argument keywords for external procedures may be specified in a procedure interface block (12.3.2.1).

NOTE 2.5

> Argument keywords can make procedure references more readable and allow actual arguments to be in any order. This latter property facilitates use of optional arguments.

2.5.3 Declaration

The term **declaration** refers to the specification of attributes for various program entities. Often this involves specifying the data type of a named data object or specifying the shape of a named array object.

2.5.4 Definition

The term **definition** is used in two ways. First, when a data object is given a valid value during program execution, it is said to become **defined**. This is often accomplished by execution of an assignment statement or input statement. Under certain circumstances, a variable does not have a predictable value and is said to be **undefined**. Section 14 describes the ways in which variables may become defined and undefined. The second use of the term **definition** refers to the declaration of derived types and procedures.

2.5.5 Reference

A **data object reference** is the appearance of the data object name or data subobject designator in a context requiring its value at that point during execution.

A **procedure reference** is the appearance of the procedure name or its operator symbol or the assignment symbol in a context requiring execution of the procedure at that point.

The appearance of a data object name, data subobject designator, or procedure name in an actual argument list does not constitute a reference to that data object, data subobject, or procedure unless such a reference is necessary to complete the specification of the actual argument.

A **module reference** is the appearance of a module name in a USE statement (11.3.1).

2.5.6 Association

Association may be name association (14.6.1), pointer association (14.6.2), or storage association (14.6.3). Name association may be argument association, host association, or use association.

Storage association causes different entities to use the same storage. Any association permits an entity to be identified by different names in the same scoping unit or by the same name or different names in different scoping units.

2.5.7 Intrinsic

The qualifier **intrinsic** signifies that the term to which it is applied is defined in this standard. Intrinsic applies to data types, procedures, assignment statements, and operators. All intrinsic data types, procedures, and operators may be used in any scoping unit without further definition or specification.

2.5.8 Operator

An **operator** specifies a particular computation involving one (unary operator) or two (binary operator) data values (**operands**). Fortran contains a number of intrinsic operators (e.g., the arithmetic operators +, –, *, /, and ** with numeric operands and the logical operators .AND., .OR., etc. with logical operands). Additional operators may be defined within a program (7.1.3).

2.5.9 Sequence

A **sequence** is a set ordered by a one-to-one correspondence with the numbers 1, 2, through n. The number of elements in the sequence is n. A sequence may be empty, in which case it contains no elements.

The elements of a nonempty sequence are referred to as the first element, second element, etc. The nth element, where n is the number of elements in the sequence, is called the last element. An empty sequence has no first or last element.